UNIVERSIDAD YACHAY TECH

ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES

Containerized Deployment of a Flask + MySQL Application Using Docker Compose and Kubernetes (Minikube)

Student Name: Daniel Troya A.

Instructor Name: PhD. Rolando Armas

Course : Management of Computing Services

Date: June 3, 2025

School of Mathematical and Computational Sciences

Contents

1 Executive Summary

In this project, we will implement a MySQL database along with its corresponding APIs. Additionally, we will deploy a solution that reinforces containerization and enhances how services are exposed and scaled through the use of Kubernetes. This setup ensures better portability, easier deployment, and automated management of resources. It also allows the system to handle higher loads efficiently by leveraging Kubernetes' orchestration capabilities.

1.1 Objectives

- Understand the design and management of databases and APIs to support the implementation and improvement of the service.
- Design APIs that align with the overall system architecture, ensuring proper functionality and effective communication with the database.
- Understand the purpose and advantages of using Kubernetes, and how it can improve the scalability and reliability of the system in real-world scenarios.

1.2 Theorical Framework

Docker

Docker is software that allows us to package and run applications within an isolated environment called a *container*. These containers are separated from the host operating system, making it easier to run multiple services or applications without interference between them. One of its main advantages is that it enables the creation of customized environments quickly and efficiently.

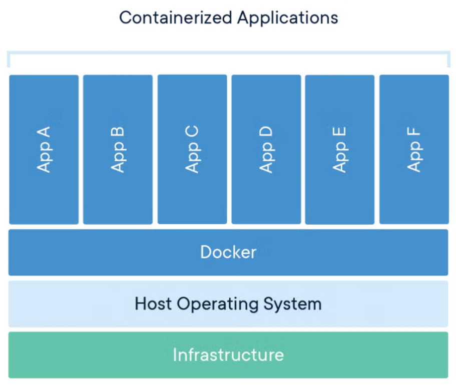


Figure 1: Docker: Graphical Example

Kompose
Kompose is a conversion tool that helps automate the containerization of Docker-based applications for deployment in Kubernetes. It allows you to specify images, ports, and services defined in a Docker Compose file and convert them into Kubernetes manifests.

With this tool, we can simplify the orchestration process by generating the necessary YAML or JSON configuration files (such as Deployments, Services, or DaemonSets) to ensure correct functionality within a Kubernetes environment.

Flask

Flask is a lightweight application framework designed for creating APIs that facilitate communication between different applications. This framework enhances the efficiency of system development by providing a simple and flexible structure for building web services.

In this project, we use the Flask library in Python to simplify the creation of the system architecture and manage the functionality of each API effectively.

Figure 2: Flask: Graphical API Connectivity Example

MySQL

MySQL is an open-source relational database management system that allows us to efficiently store, organize, and manage data. It is one of the most widely used relational databases today.

MySQL uses Structured Query Language (SQL), which enables us to interact with the database, perform searches, and retrieve specific information with precision. It is considered relational because it uses relationships between tables to avoid data duplication, allowing entities to be referenced rather than repeated.

Kubernetes

Kubernetes is an open-source container orchestration system that helps automate software deployment, scaling, and management. Its main goal is to group one or more computers—whether virtual machines (VMs) or physical servers—into a unified cluster.

This allows you to test applications locally or run them in large-scale enterprise environments. Kubernetes offers flexibility and scalability, enabling you to deliver your applications consistently and efficiently, regardless of the complexity of your infrastructure or requirements.

Minikube

Minikube is a local version of a Kubernetes environment that runs directly on your computer. By default, it uses a single-node cluster, making it ideal for development and testing purposes.

When starting Minikube, it sets up a Kubernetes cluster on your machine. Depending on your configuration, it can run in the following environments:

- Inside a Docker container
- Inside a virtual machine (VM)
- Directly on your local machine

2 System Architecture

The API layer enables communication between the user interface and the MySQL database. Using the Marshmallow library, we define schemas that handle automatic data validation and facilitate the transformation between JSON data and Python objects.

To ensure proper communication, we specify in the main script the URL used to connect from other applications or programs to the container. Typically, this follows the format: **http://localhost:port**.

Maintain a modular and scalable structure, each table in the database has its own Python script implementing its respective CRUD operations. These scripts are organized using Flask `Blueprints`, which allow us to separate logic by module and import them all into a single entry point:

- `use_db.py:` This script centralizes the SQLAlchemy `db` instance, avoiding redundancy in each module.
- `main.py:` The main application file where all Blueprints are registered. It also defines the host and port `3306` for connecting to the MySQL container.

For more information, the folder `flask_api` contains:

- **flaskenv:** Environment configuration file containing all required libraries.

- **Python scripts (.py):** All modular CRUD APIs developed for each database table.

The diagra used was:

3 Docker-Based Implementation

All scripts follow a consistent structure composed of the following sections:

Header

This section includes the import statements for essential libraries such as `Flask`, `Blueprint`, `SQLAlchemy`, and `Marshmallow`.

- Each script defines its own `Blueprint` to modularize the API routes.
- These Blueprints are then imported and registered in `main.py` to activate the routes.

Model

The model defines a SQLAlchemy class that directly maps to a table in the MySQL database.

- Each class attribute corresponds to a column in the table.
- Data types, primary keys, and constraints such as `nullable` and `unique` must be explicitly defined.

Marshmallow Schema

This section includes a Marshmallow schema class used for serializing and deserializing Python objects to and from JSON.

- It ensures that all incoming and outgoing data is validated before interacting with the database.
- This feature also improves compatibility with front-end frameworks like React, Vue, or REST clients.

Endpoints (CRUD Operations)

Each script defines multiple Flask routes to handle CRUD (Create, Read, Update, Delete) operations.

- These routes use the previously defined model and schema to process requests.
- All route handlers are exposed via `main.py` by registering the corresponding Blueprints.

3.1 Dockerfile Settings

Here we can observe the `Dockerfile` implementation based on the `python:3.10-slim` image.

```
FROM python:3.10-slim

# Install build dependencies required for MySQL/MariaDB
    support
RUN apt-get update && apt-get install -y netcat-openbsd gcc
    libmariadb-dev \
     build-essential \
     libssl-dev \
     libffi-dev \
     python3-dev \
     && rm -rf /var/lib/apt/lists/*

# Set the working directory inside the container
WORKDIR /app

# Copy and install Python dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy a script that wait for services like the database
COPY wait-for-it.sh ./wait-for-it.sh
RUN chmod +x wait-for-it.sh

# Copy the rest of the application code
COPY . .

# Default command to run the application
CMD ["python", "app/main.py"]
```

3.2 Environment Variables

Before creating the `docker-compose.yml` file, we must define the environment variables used by our database. Docker Compose will automatically load these variables from the `.env` file and apply them during container initialization.

The variables we define include the database name, user credentials, and connection details. This allows us to securely and easily configure the database container.

```
MYSQL_ROOT_PASSWORD=ldtroya04
MYSQL_DATABASE=final_db
MYSQL_USER=user
MYSQL_PASSWORD=ldtroya04
MYSQL_HOST=db
MYSQL_PORT=3306
```

3.3 Docker Compose YAML

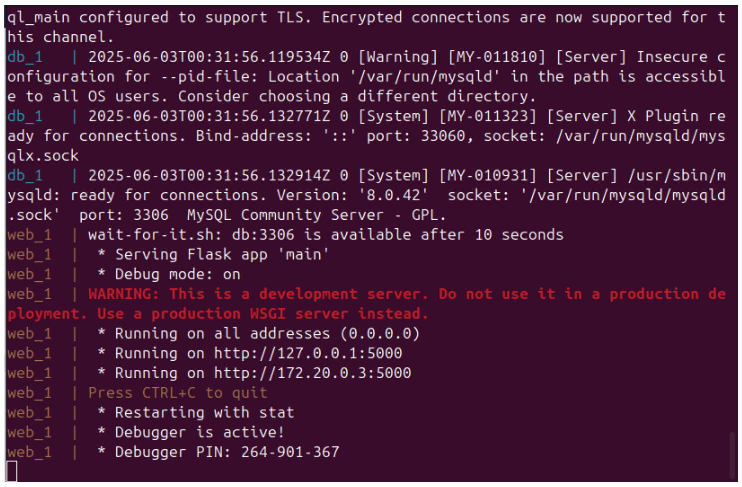
To configure the YAML file, we need to specify the names of the images we are going to use, as well as correctly set the ports. It's important to be careful when assigning ports—using the wrong port may lead to connection errors.

We also created a `.sh` script that allows the Python API to wait until the database server is fully running, helping to avoid startup errors due to service timing.

In addition, we use volumes to directly import the database into the container and to persist information, such as data generated by the services or database, across container restarts.

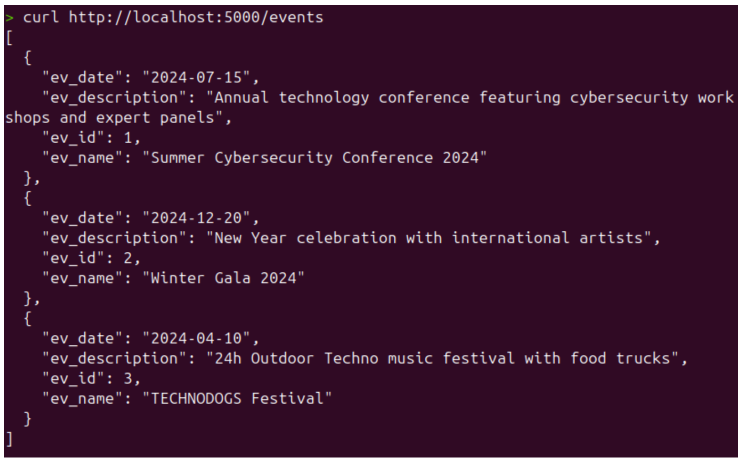
```
services:
  db:
    image: mysql:8.0  # Image
    env_file:
      - ./env         # Load environment variables
    ports:
      - "3307:3306"   # Define Ports
    volumes:
      - ./mysql_data:/var/lib/mysql
      - ./app/init.sql:/docker-entrypoint-initdb.d/init.sql
    restart: always    # Always restart the container if it
       stops

  web:
    build: .           # Build the image with Dockerfile
    command: ["./wait-for-it.sh", "db:3306", "--", "python",
         "app/main.py"]
                        # Wait for the DB to be ready, then
                           run the main app
    volumes:
      - .:/usr/src/app
    ports:
      - "5000:5000"    # Define Ports
    depends_on:
      - db
    env_file:
      - ./env          # Load environment variables for the
         app
```

The results was:



```
ql_main configured to support TLS. Encrypted connections are now supported for t
his channel.
db_1   | 2025-06-03T00:31:56.119534Z 0 [Warning] [MY-011810] [Server] Insecure c
onfiguration for --pid-file: Location '/var/run/mysqld' in the path is accessibl
e to all OS users. Consider choosing a different directory.
db_1   | 2025-06-03T00:31:56.132771Z 0 [System] [MY-011323] [Server] X Plugin re
ady for connections. Bind-address: '::' port: 33060, socket: /var/run/mysqld/mys
qlx.sock
db_1   | 2025-06-03T00:31:56.132914Z 0 [System] [MY-010931] [Server] /usr/sbin/m
ysqld: ready for connections. Version: '8.0.42'  socket: '/var/run/mysqld/mysqld
.sock'  port: 3306  MySQL Community Server - GPL.
web_1  | wait-for-it.sh: db:3306 is available after 10 seconds
web_1  |  * Serving Flask app 'main'
web_1  |  * Debug mode: on
web_1  | WARNING: This is a development server. Do not use it in a production de
ployment. Use a production WSGI server instead.
web_1  |  * Running on all addresses (0.0.0.0)
web_1  |  * Running on http://127.0.0.1:5000
web_1  |  * Running on http://172.20.0.3:5000
web_1  | Press CTRL+C to quit
web_1  |  * Restarting with stat
web_1  |  * Debugger is active!
web_1  |  * Debugger PIN: 264-901-367
```

Figure 3: Docker compose process



```
> curl http://localhost:5000/events
[
  {
    "ev_date": "2024-07-15",
    "ev_description": "Annual technology conference featuring cybersecurity work
shops and expert panels",
    "ev_id": 1,
    "ev_name": "Summer Cybersecurity Conference 2024"
  },
  {
    "ev_date": "2024-12-20",
    "ev_description": "New Year celebration with international artists",
    "ev_id": 2,
    "ev_name": "Winter Gala 2024"
  },
  {
    "ev_date": "2024-04-10",
    "ev_description": "24h Outdoor Techno music festival with food trucks",
    "ev_id": 3,
    "ev_name": "TECHNODOGS Festival"
  }
]
```

Figure 4: Response from API using `curl`

4 Kubernetes

In this section, we migrate our architecture to a Kubernetes-based deployment. To do so, we must create and configure several YAML files that define our application's components.

4.1 credentials.yml

This file defines the environment variables required for database authentication. It serves the same purpose as the `.env` file described in Section (3).

To enhance security, these variables should be stored as a Kubernetes Secret. Before including them in the YAML file, the values must be encoded using Base64. This can be done in a Linux terminal with the following command:

```
$ echo -n "value" | base64
```

Once encoded, the values are placed into the `credentials.yml` file as follows:

```
# Defines the encrypted variables for database access
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
type: Opaque
# From env file
data:
  mysql-root-password: bGR0cm95YTA0    # "ldtroya04"
  mysql-user: dXNlcg==                 # "user"
  mysql-password: bGR0cm95YTA0
```

4.2 mysql-deployment.yml

This file helps us deploy the MySQL service that our Flask app will use.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql                          # Deployment name
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql                       # Remember this name
  template:
    metadata:
      labels:
        app: mysql                     # Label of pod
    spec:
      containers:
        - name: mysql                  # Container
            configurations
          image: mysql:8
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: mysql-root-password
            - name: MYSQL_DATABASE
              value: final_db
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: mysql-user
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: mysql-password
        volumeMounts:
          - name: initdb
            mountPath: /docker-entrypoint-initdb.d
    volumes:
      - name: initdb
        configMap:
          name: mysql-init
```

4.3 mysql-service.yml

We just configure the service file of mysql:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql                     # Service name
spec:
  selector:
    app: mysql                    # app name in all files
  ports:
    - port: 3306                  # Exposed port
      targetPort: 3306            # Port on the pod
```

4.4 flask-deployment.yml

This file contains all the information needed to configure and deploy the Flask application, including its connection to the rest of the system.

```
apiVersion: apps/v1                    # API version for
   Deployments
kind: Deployment
```

```
metadata:
  name: flask-api                      # Name of the
      Deployment
spec:
  replicas: 1                          # Number of pods to
      run
  selector:
    matchLabels:
      app: flask                       # Label used to match
           pods
  template:
    metadata:
      labels:
        app: flask                     # Label applied to
            the pod
    spec:
      containers:
        - name: flask                  # Name of the
            container
          image: flask-api             # Docker image to use
          imagePullPolicy: IfNotPresent # Use local image if
              available
          ports:
            - containerPort: 5000      # Port the app
                listens on
          env:
            - name: MYSQL_HOST
              value: mysql             # Hostname of the
                  MySQL service
            - name: MYSQL_PORT
              value: "3306"            # MySQL port
            - name: MYSQL_DATABASE
              value: final_db          # Database name
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: mysql-secret   # Reference to
                      Kubernetes Secret
                  key: mysql-user      # Key inside the
                      Secret
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret   # Reference to
                      Kubernetes Secret
                  key: mysql-password  # Key inside the
                      Secret
```

NOTE: We use the `imagePullPolicy` variable to ensure Kubernetes uses the locally built image. If we don't set this, we would need to upload the image to a registry to use it.

4.5 flask-service.yml

Finally, we define the Flask service file, with the purpose of exposing the Flask application and allowing external access through a specific port.

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  type: NodePort                     # Expose service on a
     port accessible from outside
  selector:
    app: flask                       # app name remember
  ports:
    - port: 5000                     # Port exposed by the
        service
      targetPort: 5000               # Port used inside the
          container
      nodePort: 30007                # External port to
          access the app
```

Additionally, we create a ConfigMap containing the SQL file. This is done because we already have a database structure and want to load it automatically into the container. To achieve this, we use the following command:

`$ kubectl create configmap mysql-init -from-file=init.sql`

This command creates a ConfigMap named `mysql-init`.

5 Deployment on Minikube

Once all the YAML configuration files have been created, we must place them in the same folder to launch the full system. Remember the architecture described in Section 2.

1. Modify the environment so Kubernetes can use local Docker images:

   ```
   $ eval $(minikube docker-env)
   ```

2. Build the Flask application image, the same way we did when using Docker Compose:

   ```
   $ docker build -t flask-api .
   ```

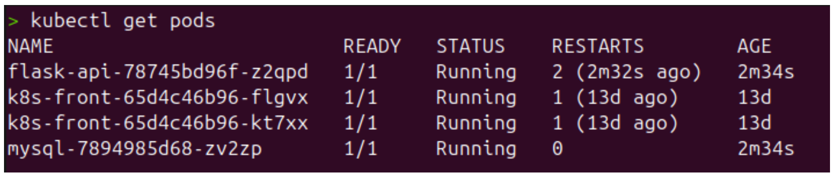
3. Apply all the configuration files using `kubectl`:

   ```
   $ kubectl apply -f mysql-deployment.yml
   $ kubectl apply -f mysql-service.yml
   ```

```
$ kubectl apply -f flask-deployment.yml
$ kubectl apply -f flask-service.yml
```

After building and deploying the system, we can verify that the pods and services:

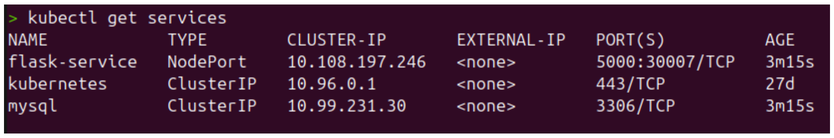
```
$ kubectl get pods
```



```
> kubectl get pods
NAME                          READY   STATUS    RESTARTS       AGE
flask-api-78745bd96f-z2qpd    1/1     Running   2 (2m32s ago)  2m34s
k8s-front-65d4c46b96-flgvx    1/1     Running   1 (13d ago)    13d
k8s-front-65d4c46b96-kt7xx    1/1     Running   1 (13d ago)    13d
mysql-7894985d68-zv2zp        1/1     Running   0              2m34s
```

Figure 5: All created pods with information about each one

```
$ curl http://192.168.49.2:30007/events
```



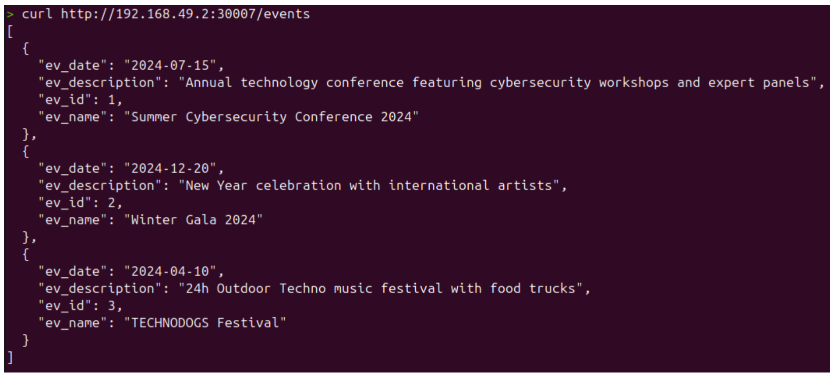
```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
flask-service   NodePort    10.108.197.246   <none>        5000:30007/TCP   3m15s
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP          27d
mysql           ClusterIP   10.99.231.30     <none>        3306/TCP         3m15s
```

Figure 6: Services Up

6 Results

For test the connection to the Flask API and database by sending a request to the service's NodePort:

```
$ curl http://192.168.49.2:30007/events
```

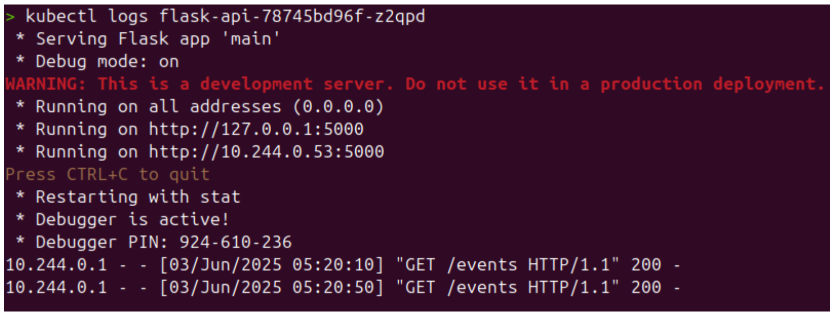


```
> curl http://192.168.49.2:30007/events
[
  {
    "ev_date": "2024-07-15",
    "ev_description": "Annual technology conference featuring cybersecurity workshops and expert panels",
    "ev_id": 1,
    "ev_name": "Summer Cybersecurity Conference 2024"
  },
  {
    "ev_date": "2024-12-20",
    "ev_description": "New Year celebration with international artists",
    "ev_id": 2,
    "ev_name": "Winter Gala 2024"
  },
  {
    "ev_date": "2024-04-10",
    "ev_description": "24h Outdoor Techno music festival with food trucks",
    "ev_id": 3,
    "ev_name": "TECHNODOGS Festival"
  }
]
```

Figure 7: Testing the connection with the database

Ans also veroify the logs on te flask-api to verify some problem

```
$ kubectl logs <pod_name>
```



```
> kubectl logs flask-api-78745bd96f-z2qpd
 * Serving Flask app 'main'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.244.0.53:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 924-610-236
10.244.0.1 - - [03/Jun/2025 05:20:10] "GET /events HTTP/1.1" 200 -
10.244.0.1 - - [03/Jun/2025 05:20:50] "GET /events HTTP/1.1" 200 -
```

Figure 8: Logs of the flask pod

7 Challenges

In this project, we faced one major challenge: implementing a pre-existing database. This problem began when we prepared the system without considering the integration of the already-created database. Initially, we mounted a new volume and created a separate YAML file with the goal of preserving the existing database columns. In that case, all registered data remained intact.

However, when we tried to integrate the existing database by modifying the YAML files, we encountered confusion due to the conflict between the old volume and the new volume managed by the Kubernetes configuration described in Section 4.

Eventually, we resolved the issue by removing the old configuration, updating the column settings in the Flask deployment, and deleting the unnecessary YAML file.

8 Conclusions

The use of Kubernetes greatly facilitates system improvement and enhances the scalability of each service. As we know, configuration files are essential to building an efficient system. It's also important to correctly define the type of database we are using. In this project, we used a pre-existing database. If you want to check it, here is the GitHub link:

`https://github.com/dtpollo/EventManagerDB.git`

In summary, Kubernetes is a powerful and useful tool for deploying our projects on the web. It helps orchestrate all the services and requirements we've defined. This is especially valuable when working on local environments for large-scale testing or more realistic setups that we can fully control.

Thanks to Docker and Flask, managing the project was much easier, as everything was separated and clearly labeled. This organization supports scalability and long-term maintenance.

Appendices

If you want to review any of the configuration files used in this project, you can find them in the main folder. Here's a brief description of each:

- **Dockerfile:** Used to build the Python container with all required dependencies.
- **Bash Script:** A shell script used to wait for the database server to be ready before starting the application.
- **YAML Files:** All Kubernetes configuration files are clearly commented for better understanding and easy deployment.
- **init.sql:** SQL file used to initialize the MySQL database. It includes table creation and some example data for managing CRUD operations.
- **/app folder:** Contains all Flask application files and API configuration.

References

- Docker Docs — Overview:
  https://docs.docker.com/get-started/docker-overview/
- Docker — What is a Container:
  https://www.docker.com/resources/what-container/
- MySQL — Understanding What It Is and How It's Used:
  https://www.oracle.com/mysql/what-is-mysql/
- Kubernetes — Official Documentation:
  https://kubernetes.io/docs/home/

- Kubernetes — ConfigMaps:
  `https://kubernetes.io/docs/concepts/configuration/configmap/`
- Kubernetes — Secrets:
  `https://kubernetes.io/docs/concepts/configuration/secret/`
- Kubernetes — Persistent Volumes: `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`
- Flask-SQLAlchemy Documentation:
  `https://flask-sqlalchemy.readthedocs.io/en/stable/`
- SQLAlchemy — MySQL Dialect Docs:
  `https://docs.sqlalchemy.org/en/20/dialects/mysql.html`
- Minikube — Pushing Images Guide:
  `https://minikube.sigs.k8s.io/docs/handbook/pushing/`